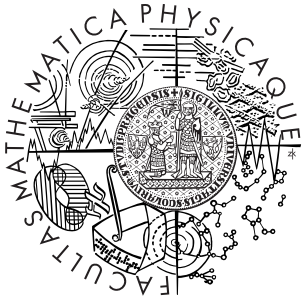
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Pavel Krč

POMDPs for dynamic troubleshooting

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Marta Vomlelová, Ph.D.
Studijní program: Informatika, obor teoretická informatika

2009

Děkuji vedoucí své práce, Mgr. Martě Vomlelové, za důkladné zasvěcení do problematiky, pomoc s výběrem literatury, inspirativní konzultace a zodpovědné vedení práce. Děkuji Doc. Ing. Emilu Pelikánovi, CSc. a Ústavu informatiky AV ČR, v.v.i., za poskytnutí výpočetní techniky a dobrého zázemí pro vývoj aplikace.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 3. 12. 2009 Pavel Krč

Contents

Název práce: POMDPs for dynamic troubleshooting
Autor: Pavel Krč
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: Mgr. Marta Vomlelová, Ph.D.
E-mail vedoucího: Marta.Vomlelova@mff.cuni.cz

Abstrakt: Pojmem dynamický troubleshooting se označuje proces analýzy běžícího systému v reálném čase, predikce a detekce možných problemů, řešení problémů a předcházení jejich výskytu. Je-li tento proces realizován počítačem, pak se ve své nejobecnější podobě jedná o problém optimálního rozhodování. Koncept částečně pozorovatelných Markovských rozhodovacích procesů (POMDP) je pro tento druh problémů velmi vhodný, nebot' umožňuje modelovat jak nejistotu ohledně budoucího vývoje procesu, tak neúplnou znalost aktuálního stavu systému a umožňuje počítat s vlastními budoucími rozhodnutími, které systém ovlivňují či přispívají k získávání informací o jeho stavu. V rámci této práce autor poskytuje úvod do teorie POMDPs a popisuje současné algoritmy řešení POMDP s přihlédnutím k jejich použitelnosti pro dynamický troubleshooting. Dále autor představuje konkrétní problém dynamického troubleshootingu, řeší jej pomocí obecných řešení POMDP a navrhuje pro něj vlastní heuristiku, která je snadno zobecnitelná i na širší třídu POMDP problémů. V programovacím jazyce Python vytváří systém pro řešení POMDP, implementuje do něj zmíněné algoritmy a testuje je na představeném problému.

Klíčová slova: Markovské rozhodovací procesy, částečná pozorovatelnost, dynamický troubleshooting, faktorizovaná reperezentace, aproximovaná optimalizace

Title: POMDPs for dynamic troubleshooting
Author: Pavel Krč
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Mgr. Marta Vomlelová, Ph.D.
Supervisor's e-mail address: Marta.Vomlelova@mff.cuni.cz

Abstract: Dynamic troubleshooting is a process of analysing a running system in real time, predicting or detecting possible problems, correcting them and acting so as to avoid them. When realised by a computer in its most generic form it is an optimum decision problem. The framework of partially observable Markov decision processes (POMDPs) is well suited for such problems as it allows modelling the uncertainty of the future evolution of the process as well as the limited knowledge about the current state and enables to presume its own future choices of actions that alter the system or gain knowledge about it. In this work the author provides an introduction to the theory of POMDPs and describes current POMDP solution algorithms with respect to their applicability for dynamic troubleshooting. Further he presents a specific dynamic troubleshooting problem, solves it using generic POMDP solutions and proposes his own heuristic for it which can be easily generalised to a wider class of POMDP problems. He creates a Python programming language framework for solving POMDPs, implements the mentioned algorithms within it and tests them on the presented problem.

Keywords: Markov decision processes, partial observability, dynamic troubleshooting, factored representation, approximate optimization

Chapter 1

Introduction

1.1 Motivation

The term *troubleshooting* represents a wide class of problem solving tasks. In general, these tasks comprise a diagnosis and analysis of a specific flaw or obstacle to a proper function of some system and suggestion (sometimes also a direct application) of an appropriate solution. More generally, troubleshooting tasks significantly or entirely participate on *optimising* the operation of a running system. Nowadays one can present a variety of specific examples from a customer support of consumer electronics to a supervision of a large factory production line.

Dynamic troubleshooting refers to systems that are rapidly evolving, so that one cannot expect the system's state to be constant during the process of analysing and eliminating the flaw. Therefore, dynamic troubleshooting methods usually operate synchronously with the target system, taking advantage of a continuous analysis and constantly updated prediction of a system's future evolution. It is clear that such a task can be very hard, especially when it is required to be near-optimal or optimal under certain conditions.

The process of troubleshooting can be purely human performed, computer assisted (expert systems) or fully automated. It is not only the cost of labour that favours the automation – the optimization nature of the problem allows computers to achieve better efficiency than human guess when it comes to exact numbers. Also, a well-tuned automatic system may be considerably less error-prone, which is often crucial for systems that are responsible for rectifying another system's errors.

As a result of that, numerous systems for dynamic troubleshooting have been developed, some of them very specific and some of them more general.

Existing systems take advantage of various theoretical backgrounds. In this work, I will focus on *Markov decision processes*, a theory that has drawn a lot of attention in artificial intelligence and operations research communities, and its application to dynamic troubleshooting.

Markov decision processes are well suitable for dynamic troubleshooting problems. They can describe any evolving system with great generality and when applied to dynamic troubleshooting, they can benefit from a good capability to describe the cause and effect in the system dynamics. I will now describe the basics of an MDP framework.

1.2 The framework in general

1.2.1 Markov decision processes

Markov process is a theoretical dynamic system, which at any time can be described with a *state*. The system evolves with time as a random process; its future states cannot be determined certainly from the current or past states, however this evolution can be described stochastically. The most important property of such system is the so-called *Markov assumption*, which says that the state holds all the information about the system at a certain time, and any future evolution of the system depends solely on that state. In terms of probability theory, any state in the future is conditionally independent on any state in the past given the current state.

Markov decision process (MDP) is a Markov system that contains an *agent*. By agent we mean an external influence to the system. At any time the agent has information about the system's state and performs actions that affect the future system's evolution. In a Markov decision process, a future state depends both on the current state and on the action that the agent has taken. Usually the agent represents our own effects on the system and is used to model our reasoning about controlling the system according to our specific needs. This reasoning can be modelled via a *policy*, which is a mapping from system's state space to the action space. The policy can be either *stationary* (i.e. constant in time) or *non-stationary*, deterministic or nondeterministic. Our goals in the system's control can be summarized in a *reward function* – a mapping either from the state space or from the Cartesian product of states and actions to (usually) real numbers. Positive numbers indicate our gain from a specific situation (being in a state or taking an action) and negative numbers mean a loss from a situation. This mapping can also be either straight or stochastic to represent uncertainty.

With all these concepts, we can formulate an optimization problem on finding a policy that maximizes the expected sum of rewards over a period of time. Solution of such a problem is then often referred to as a solution of the MDP itself. There are of course many variants on exact definitions of the presented terms as well as the target function of the optimization problem; some of these that are especially useful for dynamic troubleshooting will be presented further. Numerous algorithms and heuristics exist for solving MDPs that differ in complexity, exactness or performance, universality or applicability.

All the presented concepts (time, states, actions and rewards) can be defined either discretely or continuously. In this work I will be dealing exclusively with *full-discrete MDPs* (i.e. discrete time, states, actions and rewards). Discrete MDPs are probably most inspected by now, they are usually more tractable and for specific reasons they fit well for dynamic troubleshooting.

1.2.2 Partial observability

The MDP framework is powerful in ability to represent the dynamics of the environment (i.e. the system excluding the agent), yet it lacks an important aspect of real-world decision making problems. The problem definition assumes that at any time the agent has a complete knowledge about the environment. In reality, this is never true – both from theoretical point of view (uncertainty principle in physics) and from practical point of view (no matter how dense a network of sensors is, it never actually "sees" everywhere, also no sensor is errorless and absolutely accurate). Especially the problems where certain agent's actions can provide him with some new information cannot be accurately modelled using MDPs.

This imperfection can be overcome by adding a new concept of *observations*; the resulting framework is then called a *Partially observable Markov decision process* (POMDP). In a POMDP, the agent gains information about the environment by receiving observations that depend nondeterministically on the state of the environment and the action that the agent is performing. A single observation usually doesn't give much information about the state, yet by combining the current observation with all the past observations and actions that the agent has taken and using the knowledge of the system's dynamics, the agent might be able to determine the state of the environment quite exactly. Even more important is the fact that the agent can choose his actions according to potential new information they might bring and the expected usefulness of that information in future.

1.2.3 Factored representations

Until now we have seen a state, an action or an observation as a single value from a given (discrete or continuous) space. However, when describing real world, we usually work with multiple distinct properties. For example, when controlling a robot, an action may consist of signals sent to each motor of its body, an observation would usually be composed of several inputs from different sensors (in case of visual input we could think of individual camera pixels) and of course a state of the environment would be specified with many properties of different items that are present in that environment, including physical properties of the agent (robot) itself.

Such properties can be modelled as *variables*. After we specify the state/action/observation space for each variable, we can use some graphical techniques like *Bayesian networks* to model dependencies between the variables. Of course these variables can be transformed back into the flat state/action/observation space using Cartesian product of each variable's space. However, this so-called *localist* representation has significant drawbacks when describing any but very simple systems. Apart from being non-intuitive and complicated for exact modelling, the resulting state/action/observation space grows exponentially with the number of variables. Such a big space is able to represent literally any combination of variable assignments, not taking advantage of any potential conditional or absolute independencies. As a result of that, many problems represented that way are simply intractable.

Various algorithms that use factored representation of the problem have been proposed, both exact and approximate. The former exploit potential independencies between the variables to save both time and space while the latter simplify the structure of the model in some way (for example by considering weakly dependent variables independent, either only for some computational phases or permanently). The accuracy of some approximate representations has been intensively studied recently (e.g. [Boyen and Koller, 1998]).

1.3 General solution methods

There are two distinct approaches to finding optimal actions in an MDP framework. Either we have an exact explicit model of the process and we can solve the optimization problem to find the optimal or near-optimal stationary policy, or we don't have it and we have to learn the dynamics of the process – either from available data using supervised learning or

directly by interacting with the environment using reinforcement learning.

1.3.1 Optimization algorithms

While solving the optimization problem of an explicit model, we have to specify whether we want a *finite-* or *infinite-horizon* optimal policy. In the former case we expect the process to terminate after a predefined amount of time (i.e. time steps in discrete MDPs). This way the optimized function would usually be a simple finite sum of rewards at each time step.

In the latter case we would usually specify a *discount factor* $0 \leq \gamma < 1$ instead and we would multiply all the rewards in the future by this factor exponentially, maximizing $\sum_{t=0}^{\infty} r_t \gamma^t$ where r_t would be the reward at time t. Not only this sum is guaranteed to converge (given finite, therefore limited rewards), it would also favour earlier rewards and later losses to later rewards and earlier losses respectively, thus imitating an "interest rate". The closer to 1 we set this factor, the more the solution takes far future rewards into account, but also the slower the algorithms usually converge. The infinite sum in the optimization problem of the infinite-horizon MDP with a discount factor can be transformed to a recursive *Bellman optimality equation* (see eq. 2.3, p. 13) [Bellman, 1957].

Basic algorithms for solving both finite- and infinite-horizon MDPs and POMDPs are based on *dynamic programming* technique, starting with a simple solution for zero-length horizon and iterating recursively until the specified finite horizon or the desired accuracy for infinite horizon case is reached. (In the latter case, bounds for the error can be calculated from the maximum absolute reward and the discount factor as a sum of geometric series.)

The majority of today's algorithms follow this schema, applying various heuristics to improve the efficiency. Such an approach will be the main focus of this work as well.

1.3.2 The EM algorithm

There are lots of cases where a fully-parameterized exact model of the process isn't available. In fact, we can expect to have such a model only in theoretical problems or perhaps some very simple processes where all the physics can be exactly modelled. In most real-life problems, we only get a skeleton of the process (i.e. the sets of states, actions and observations, or variables and potential conditional independencies for factored MDPs), but the parameters (probability distributions) are not known.

If we have a sufficient amount of measured data from some model runs, we can use some machine learning method to extract the model parameters from the data. If the data contain all the information necessary to describe the process, this is simply a matter of statistics. If, on the other hand, some information is hidden (typically states for POMDPs), we have to estimate them first.

The *expectation-maximization (EM) algorithm* [Dempster et al., 1977] is a generic solution to this problem. Suppose we have a model with unknown parameters, some initial guess of these parameters and the data that are missing some latent (hidden) variables, the algorithm consists of two steps that are repeated iteratively:

Expectation (E) step: Using the current model parameters, estimate the (distribution over the) hidden variables.

Maximization (M) step: Find such parameters of the model that explain the data (available plus those estimated in the E-step) with maximum likelihood. Set these parameters as current.

It can be shown that when these two steps are iterated, the likelihood will increase or stop at a local maximum [Sallans, 2002]. If finding the maximum-likelihood parameters in the M-step is too hard, it can be substituted by a partial M-step that only finds parameters that improve the likelihood. After the model parameters have been found, the problem can be solved using algorithms mentioned in 1.3.1.

1.3.3 Reinforcement learning

In case we have neither full model parameterization nor data from model runs, we are facing a learning problem similar to learning in biology. The agent must be placed in the environment and learn from the responses to its own actions. It should have two main goals: to learn the dynamics of the environment and to gain as much reward as possible, the former being just means for the latter. Therefore it also solves the *exploration vs. exploitation* dilemma, i.e. the matter of how much effort should one put into exploring new "areas" (states) and improving the knowledge vs. gaining maximum reward using already available knowledge throughout the learning process [Kaelbling et al., 1996].

As the agent interacts with the environment, it updates its estimate of model parameters, hopefully converging to the true values. A class of learning algorithms called *temporal difference (TD) learning* doesn't require that the agent remembers a full history of its actions. Instead, it

remembers only the current estimate of model parameters (or a function of these), that is shifted after every step towards the current response from the environment using a decreasing *learning rate*. A typical example of these are the *Q-learning* [Watkins, 1989] and the *SARSA* [Sutton, 1996] algorithms. Neither of these needs to have a full model of the environment at any time, thus reducing space complexity.

Although these algorithms usually converge much slower than the exact model optimization algorithms, there are problems which are intractable for the exact algorithms due to time or space complexity but which can be successfully approximated using reinforcement learning.

1.4 Aims and organization of this work

This work has two main goals. The first one is to summarize the available knowledge about POMDP theory and solution techniques with respect to applicability in dynamic troubleshooting and to provide them in a form of a comprehensible introduction. This introduction forms part 1 of this work (chapters 2 and 3).

Chapter 2 provides a basic introduction to fully observable Markov decision processes and describes two standard algorithms used to solve them. It serves as ground for explaining the more complex framework of POMDPs. This framework is further explained in chapter 3. It starts with a rigorous definition of the problem, some other theoretical concepts that are connected with it and explains a general, purely theoretical approach to solving problems. Finally it describes two important real algorithms used for finding exact solution of general POMDP problems.

The second part of this book (chapters 4–7) is devoted to the second main goal of this work: to take a specific dynamic troubleshooting problem, formalise it within POMDP, search for applicable generic solution approaches and possibly propose an own solution approach usable for the specific problem or even more generally.

Chapter 4 defines the problem within POMDP formalism and explains the only one solution know to the author so far. Chapter 5 outlines an original approach that might be used for this problem as well as a wider class of POMDP problems. Chapter 6 briefly describes the programming part of this work, which includes creating a framework for both solving and simulating certain type of POMDP problems including the mentioned one, full implementation of two generic solution algorithms from scratch and two implementations of the proposed heuristic on a mentioned POMDP problem. Chapter 7 summarizes some of the results

that were obtained by testing this software with various configurations and discusses the contribution of the proposed heuristic and its potential for further enhancements.

Part I

A survey on existing methods

Chapter 2

Fully observable MDPs

Note In this and the following chapter I will be dealing with exact definitions of some concepts. Unfortunately, some of these concepts vary in definition from paper to paper, although the problems are usually reducible from one representation to another with no effect on its asymptotic complexity. I have tried to always choose a well-known definition, referencing the source of it. In some cases where there are other important definitions, I only mention them without using them further.

2.1 Theoretical background

According to [Cassandra et al., 1994], a *Markov decision process* is a tuple $(\mathcal{S}, \mathcal{A}, T, R)$ where

$\mathcal{S}$ is a finite set of states,

$\mathcal{A}$ is a finite set of actions,

$T : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{S})$ is a transition model where $T(s, a, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ [1] is the probability that the state changes to s' from s after the agent takes action a,

$R : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is a reward (positive values) / cost (negative values) function.

($\Delta(\mathcal{S}) = \{(p_s)_{s\in\mathcal{S}} : 0 \le p_s \le 1, \sum_{s\in\mathcal{S}} p_s = 1\}$ is the $\mathcal{S}$-simplex.)

The transition model T is sometimes presented as an $|\mathcal{S}| \times |\mathcal{S}|$ matrix T_a for each action a, which comes in useful later with the POMDPs. The

[1] For simplicity, I write $T(s, a, s')$ instead of the rigorous notation $\big(T(s, a)\big)(s')$

reward function is sometimes defined as $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$, making the rewards dependent also on the state after the agent's action, or – on the contrary – just as $R : \mathcal{S} \mapsto \mathbb{R}$. The rewards may also be defined stochastically: $\Pr(r_t = x \mid s_t = s, a_t = a) = p_x$, however this mapping can safely be simplified to the expected value $\sum_x p_x x$ without affecting the reasoning, because the agent wouldn't be able to influence what reward it would obtain given a single action and would therefore consider this *expected immediate reward* anyway if aiming for the maximum expected total reward.

For a problem defined this way, the Markov assumption is valid – the next state of the process depends solely on the previous state and the agent's action. Therefore, only the knowledge of the current state is required for (optimal) reasoning.

A stationary policy $\pi : \mathcal{S} \mapsto \mathcal{A}$ is a direct mapping from states to actions, a non-stationary policy is time-dependent. It is clear that for a finite-horizon MDP, the optimal reasoning depends on the time left before the process stops – for example if there were a state with a big reward that could be reached from the current state only via a long sequence of other states, the optimal agent would choose actions leading to this state only if there were enough time left to reach it.

However, for the infinite-horizon MDPs with a discount factor, there is always an *optimal stationary policy* π^* that maximizes the expected discounted reward [Howard, 1960]. In the next section I present some well-known algorithms for finding such a policy.

2.2 Dynamic programming algorithms

Given a stationary policy π, we have a regular Markov process defined as follows:

$$\begin{aligned}\Pr_{MP}(s_{t+1} = s' \mid s_t = s) &= \Pr_{MDP}\big(s_{t+1} = s' \,\big|\, s_t = s, a_t = \pi(s_t)\big) \\ &= T\big(s, \pi(s), s'\big)\end{aligned}$$

The probabilities defined for the Markov decision process are marked $_{MDP}$ while the newly defined probabilities for the Markov process are marked $_{MP}$. Technically, because Markov processes have no concept of rewards, we may rather declare the state to be a tuple (state, reward), while the next state depends only on the first element:

$$\Pr_{MP}\big(s_{t+1} = (s', r') \,\big|\, s_t = (s, r)\big) = \begin{cases} T\big(s, \pi(s), s'\big) & \text{for } r' = R\big(s, \pi(s)\big) \\ 0 & \text{for any other } r' \end{cases}$$

For this process we can calculate the *expected discounted reward* when starting from state s:

$$\begin{aligned}
V^\pi(s) &= \mathrm{E}\left\{ \sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, \begin{array}{l} s_0 = s, \\ a_t = \pi(s_t) \end{array} \right\} \\
&= \mathrm{E}\left\{ r_0 + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} r_t \,\middle|\, \begin{array}{l} s_0 = s, \\ a_t = \pi(s_t) \end{array} \right\} \\
&= R\big(s, \pi(s)\big) + \gamma\, \mathrm{E}\left\{ \sum_{t=1}^{\infty} \gamma^{t-1} r_t \,\middle|\, \begin{array}{l} s_0 = s, \\ a_t = \pi(s_t) \end{array} \right\} \\
&= R\big(s, \pi(s)\big) + \gamma \sum_{s' \in \mathcal{S}} T\big(s, \pi(s), s'\big)\, \mathrm{E}\left\{ \sum_{t=1}^{\infty} \gamma^{t-1} r_t \,\middle|\, \begin{array}{l} s_1 = s', \\ a_t = \pi(s_t) \end{array} \right\} \\
&= R\big(s, \pi(s)\big) + \gamma \sum_{s' \in \mathcal{S}} T\big(s, \pi(s), s'\big) V^\pi(s')
\end{aligned} \tag{2.1}$$

These recursive equations (one for each $s \in \mathcal{S}$) are called the *Bellman equations for policy* π and the function V^π is called the *value function* for policy π.

Due to the recursive nature of this equation, the value function can also be computed recursively – starting from some initial value V_0^π (usually $V_0^\pi(\cdot) = 0$) we can update the value function as follows:

$$\forall s \in \mathcal{S} : V_{k+1}^\pi(s) = R\big(s, \pi(s)\big) + \gamma \sum_{s' \in \mathcal{S}} T\big(s, \pi(s), s'\big) V_k^\pi(s') \tag{2.2}$$

A single update is called a *full backup* since we are updating the values for each state. The sequence $V_k^\pi(s)$ will converge to the correct value of $V^\pi(s)$ with bounds given by the sum of the geometric series $\sum_{i=k}^{\infty} \gamma^i \max_{s,a} |R(s,a)|$.

Given that $V_0^\pi(\cdot) = 0$, the elements of the sequence V_k^π are in fact value functions for the discounted k-horizon MDP with policy π; setting $\gamma = 1$ makes this regular finite-horizon MDP.

Getting back to infinite-horizon MDPs, the optimal policy π^* will always choose the action that maximizes the expected discounted reward,

therefore for the *optimal value function* we have:

$$\begin{aligned} V^*(s) &= \mathrm{E}\left\{ \sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, \begin{array}{l} s_0 = s, \\ a_t = \pi^*(s_t) \end{array} \right\} \\ &= \max_{a \in \mathcal{A}} \mathrm{E}\left\{ r_0 + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} r_t \,\middle|\, \begin{array}{l} s_0 = s, \\ a_0 = a, \\ \forall t > 0 : a_t = \pi(s_t) \end{array} \right\} \\ &= \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V^*(s') \right\} \end{aligned} \tag{2.3}$$

These $|\mathcal{S}|$ equations are called the *Bellman optimality equations* and the result V^* is the *optimal value function*. Having found this, our problem is solved, because we can simply set

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V^*(s') \right\} \tag{2.4}$$

and we have the optimal stationary policy.

2.2.1 Value iteration

This leads us to the first MDP solution algorithm called *value iteration* [Bellman, 1957]. We can compute the optimal value function similarly as in eq. 2.2:

$$\forall s \in \mathcal{S} : V^*_{k+1} = \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V^*_k(s') \right\} \tag{2.5}$$

The whole algorithm then goes as follows:

1. Set $V^*_0(\cdot) = 0$.
2. Repeat eq. 2.5 for $k = 0, 1, \ldots$ until $\forall s \in \mathcal{S} : |V^*_{k+1}(s) - V^*_k(s)| < \epsilon$.
3. Set $V^* = V^*_k$ (for the last used k).
4. Set π^* according to eq. 2.4.

This algorithm is guaranteed to converge in finite number of steps and the resulting policy is within $\frac{2\gamma\epsilon}{1-\gamma}$ from the true optimal policy [Lovejoy, 1991b].

As it was for the generic policy, the intermediate value functions V_k^* are in fact optimal value functions for the discounted k-horizon MDP (i.e. the k-horizon value functions for the k-horizon optimal policy, not the k-horizon value functions for the infinite-horizon optimal policy π^*) and so is in fact the resulting policy. A modification of this algorithm with a predefined number of iterations and $\gamma = 1$ can therefore be used to solve a finite-horizon MDP.

2.2.2 Policy iteration

Howard [Howard, 1960] proposed a slightly different algorithm called *policy iteration* or *Howard's routine* based on improving a policy until the optimal one is found:

1. Choose an arbitrary policy π_0.
2. Repeat for $k = 0, 1, \ldots$:
   (a) Calculate V^{π_k} either by repeating eq. 2.2 until it converges or by solving eq. 2.1 as a set of linear equations.
   (b) Improve the policy by choosing optimal actions according to value function of the current policy:

$$\pi_{k+1}(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V^{\pi_k}(s') \right\}$$

   until $\pi_{k+1} = \pi_k$.
3. Set $\pi^* = \pi_k$ (for the last used k).

Step 2(b) is guaranteed to improve the policy if it is not already optimal, therefore the algorithm will stop after a finite number of steps and will find the optimal policy as long as step 2(a) computes the value function exactly [Howard, 1960].

This algorithm has the advantage that when step 2(a) is computed using linear equations, there is a definite stopping condition. As with the value iteration, it can be used to solve the finite-horizon MDPs as well after a proper modification of step 2(a) and when $\gamma = 1$.

Comparing the value iteration and policy iteration algorithms according to temporal complexity, neither one of them is definitely better. It has been proven that the policy iteration algorithm requires at most the same number of steps as value iteration when considering the whole step

2 of it a single step [Puterman, 1994]. However, as a single iteration of step 2(a) in policy iteration may (and usually will) take more time than a single iteration of step 2 in value iteration, it is not sure which algorithm will perform better on a given problem.

There is also a combination of these two algorithms called the *modified policy iteration* [Puterman and Shin, 1978] that may perform better than these two on some problems. This algorithm is similar to policy iteration except that step 2(a) is computed via eq. 2.2 repeated only several times, not waiting until it converges.

Chapter 3

Partially observable MDPs

As was mentioned in 1.2.2, adding the principle of only partial observability of the world greatly extends the class of real-world problems that can be modelled in our framework. From the available ways to theoretically model the partial or fuzzy observability of some hidden state, the POMDP framework uses probably the most general model of stochastic *observations*: from a continuous or discrete set of observations the agent receives a single observation for each time step or time point. Which observation the agent receives depends stochastically on the current state and is conditionally independent on all the past states and observations given the current state, thus preserving the Markov property.

The probability distributions for receiving observations are specified in the form $\Pr(observation|state)$, not vice versa, following the true cause–effect order in most real-world problems. However, as the agent receives observations and needs to reason probabilistically about states, it must transform this knowledge using the Bayes' theorem, which brings requirement of the *initial probability distribution of state* of the environment at the beginning of the process. As the process evolves and the agent receives more and more observations, the initial distribution becomes less significant in the current guess of the state, therefore for a long-lasting or highly observable processes, the initial distribution can be given arbitrarily as a uniform or random distribution. Where the initial distribution is important, it can be inferred theoretically or estimated statistically like other parameters of the process. It can also be learnt from the observation data as a maximum likelihood estimate when the other parameters are known.

Similarly as with the previously introduced concepts, I will focus on finite discrete observation sets in the rest of this work.

3.1 Extending the MDP framework with partial observability

In case of POMDPs, the exact definitions and nomenclature vary even more than with the MDPs. I will extend our definition of MDPs from 2.1 same way as in [Russell and Norvig, 2003]. A *Partially observable Markov decision process* is given by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, T, O, R, b_0)$:

$\mathcal{S}$ is a finite set of states,

$\mathcal{A}$ is a finite set of actions,

$\mathcal{O}$ is a finite set of observations,

$T : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{S})$ is a transition model as in 2.1,

$O : \mathcal{S} \mapsto \Delta(\mathcal{O})$ is an observation model where $O(s, o) = \Pr(o_t = o \mid s_t = s)$ [1] is the probability of getting observation o while being in state s,

$R : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is a reward function and

$b_0 \in \Delta(\mathcal{S})$ is an initial state distribution.

Various alternate definitions and notations for T and R have been shown in 2.1; they apply to POMDPs as well. The observation model can be made dependent also on actions and/or on both *prior* and *posterior* state: $O : \mathcal{S} \times \mathcal{A} \mapsto \Delta(\mathcal{O})$ where $O(s, a, o) = \Pr(o_{t+1} = o \mid s_t = s, a_t = t)$ or $O : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \Delta(\mathcal{O})$ where $O(s, a, s', o) = \Pr(o_{t+1} = o \mid s_t = s, a_t = a, s_{t+1} = s')$ respectively. Neither of this would mean a fundamental change to the framework or its tractability.

3.1.1 Belief state

As was mentioned earlier, the agent needs some probabilistic reasoning about the possibility of being in a specific state. With the observation history, it's own past action choices and the initial state distribution being the only agent's information about the state, it is clear that the best estimate would be simply

$$\forall s \in \mathcal{S}, t \geq 1 : b_t(s) \stackrel{\text{def}}{=} \Pr(s_t = s \mid b_0, a_0 \ldots a_{t-1}, o_1 \ldots o_t)$$

[1] $O(s, o)$ stands for $\big(O(s)\big)(o)$

This probability distribution $b_t \in \Delta(\mathcal{S})$ is called a *belief state*. The initial state distribution b_0 is sometimes called the *initial belief state* for it is clearly identical to belief state at time 0, so this nomenclature b_0 vs. b_t makes sense.

The fundamental fact about POMDPs is that the belief state *entirely summarizes* all the agent's knowledge about the state. That is, given a belief state b_t, subsequent action a_t and resultant observation o_{t+1}, the new belief state b_{t+1} can be calculated without the knowledge of any actions, observations or belief states further in history. This process is called a *belief update* and is in fact a single step of *recursive estimation* in filtering of *hidden Markov models* [Russell and Norvig, 2003]. The probabilities can be calculated straightforwardly using Bayes' rule:

$$\begin{aligned} b_{t+1}(s') &= \Pr(s_{t+1}=s' \,|\, a_t=a, o_{t+1}=o, b_t) \\ &= \frac{\Pr(o_{t+1}=o \,|\, s_{t+1}=s', a_t=a, b_t)\Pr(s_{t+1}=s' \,|\, a_t=a, b_t)}{\Pr(o_{t+1}=o \,|\, a_t=a, b_t)} \\ &= \frac{O(s',o)\sum_{s\in\mathcal{S}} T(s,a,s')b_t(s)}{z(b_t,a,o)} \end{aligned}$$

where

$$z(b_t,a,o) = \Pr(o_{t+1}=o \,|\, a_t=a, b_t) = \sum_{s'\in\mathcal{S}} O(s',o)\sum_{s\in\mathcal{S}} T(s,a,s')b_t(s)$$

As the new belief state is calculated completely (for each s'), the computation of the denominator $z(b_t,a,o)$ can be avoided by simply normalizing the result. If the transition model is given as $|\mathcal{A}|$ matrices T_a where $[T_a]_{s's} = T(s,a,s')$ and we have a diagonal matrix $[O_o]_{ss} = O(s,o)$ for each $o \in \mathcal{O}$, we can define a new operator BU that performs the belief update (b_t is a column vector):

$$b_{t+1} = BU(b_t,a,o) = \frac{O_o T_a b_t}{z(b_t,a,o)} = \frac{O_o T_a b_t}{\|O_o T_a b_t\|} \tag{3.1}$$

3.1.2 Belief MDP

As the process is running, the agent maintains a single belief state, starting with b_0 and updating via BU, and chooses his actions according to a policy, which is defined for belief states ($\pi : \Delta(\mathcal{S}) \mapsto \mathcal{A}$).

Substituting states with belief states, this whole process can be seen as a fully observable Markov process above the continuous state space $\Delta(\mathcal{S})$. The transition model is derived from the belief update, having observations

as the nondeterministic factor, i.e. the probability of transition from belief state b to b' is the sum of probabilities of observations that would cause such a belief update or zero if there is no such observation:

$$\begin{aligned} T_{MDP}(b_t, a, b_{t+1}) &\stackrel{\text{def}}{=} \tau(b_t, a, b_{t+1}) \\ &\stackrel{\text{def}}{=} \sum_{\substack{o \in \mathcal{O} \\ BU(b_t,a,o)=b_{t+1}}} \Pr(o_{t+1} = o \mid a_t = a, b_t) \\ &= \sum_{\substack{o \in \mathcal{O} \\ BU(b_t,a,o)=b_{t+1}}} z(b_t, a, o) \end{aligned}$$

The rewards are defined as expected rewards from the POMDP:

$$R_{MDP}(b, a) \stackrel{\text{def}}{=} \rho(b, a) \stackrel{\text{def}}{=} \sum_{s \in \mathcal{S}} b(s) R_{POMDP}(s, a)$$

This process is indeed Markov, as the new belief state is conditionally independent on the past belief states given the current belief state and action. We call it the *belief process* (belief MDP) and the original POMDP is then called the *core process*. It can be shown that if the agent follows the optimal policy for the belief process, it also gains maximum expected reward from the partially observable point of view. We have therefore reduced the problem of solving POMDPs to solving continuous-state MDPs. However, this brings a new problem of representing the continuous, potentially high-dimensional state space.

3.1.3 Value function and value iteration

Because the policy in POMDPs is defined for belief states, so has to be the value function. Using definition of the belief process, the Bellman policy equations for POMDPs are:

$$\begin{aligned} V^\pi(b) &= \rho(b, \pi(b)) + \gamma \sum_{b' \in \Delta(\mathcal{S})} \tau(b, \pi(b), b') V^\pi(b') \\ &= \rho(b, \pi(b)) + \gamma \sum_{b' \in \Delta(\mathcal{S})} \sum_{\substack{o \in \mathcal{O} \\ BU(b,\pi(b),o)=b'}} z\big(b, \pi(b), o\big) V^*(b') \\ &= \rho(b, \pi(b)) + \gamma \sum_{o \in \mathcal{O}} z\big(b, \pi(b), o\big) V^\pi\Big(BU\big(b, \pi(b), o\big)\Big) \end{aligned}$$

and the Bellman optimality equations are:

$$\begin{aligned} V^*(b) &= \max_{a\in\mathcal{A}} \left\{ \rho(b,a) + \gamma \sum_{b'\in\Delta(\mathcal{S})} \tau(b,a,b') V^*(b') \right\} \\ &= \max_{a\in\mathcal{A}} \left\{ \rho(b,a) + \gamma \sum_{o\in\mathcal{O}} z(b,a,o) V^*\big(BU(b,a,o)\big) \right\} \end{aligned} \tag{3.2}$$

with the recursive computation step:

$$\forall b \in \Delta(\mathcal{S}) :$$

$$V^*_{k+1}(b) = \max_{a\in\mathcal{A}} \left\{ \rho(b,a) + \gamma \sum_{o\in\mathcal{O}} z(b,a,o) V^*_k\big(BU(b,a,o)\big) \right\} \tag{3.3}$$

There is a of representing a value function of uncountable domain $\Delta(\mathcal{S}) \mapsto \mathbb{R}$. Although nowadays there are many simple or sophisticated ways of approximating it (linear or polynomial approximation, various grids with interpolation, neural networks), the basic representation exploits an important fact: *The optimal value function for a finite-horizon discounted or undiscounted POMDP is piecewise linear and convex* (PWLC) [Sondik, 1971]. Indeed, it is a result of finitely many applications of linear operators and maximization (eq. 3.3). It can therefore be represented as

$$V^*_k(b) = \max_{\alpha\in\mathcal{V}_k} b \cdot \alpha \tag{3.4}$$

where $\mathcal{V}_k$ is a set of $|\mathcal{S}|$-dimensional vectors.

For the infinite horizon case, the optimal value function is no longer guaranteed to be PWLC, however it can be approximated with a PWLC function to an arbitrary degree of accuracy, using a finite horizon long enough [Sondik, 1971]. There is also a subclass of POMDPs called *finitely transient* POMDPs for which the optimal value function is PWLC even in the infinite-horizon case [Sondik, 1978].

3.2 Exact POMDP algorithms

In this section I will present two well-known algorithms for solving POMDPs, both of which use vector sets to represent the value function. They are therefore guaranteed to be exact only for finite-horizon problems and arbitrarily close to exact solution for infinite-horizon problems; nevertheless this is what is usually meant by the term “exact POMDP solution algorithm”.

3.2.1 Exhaustive Enumeration

This algorithm, introduced by Monahan [Monahan, 1982], was neither the first POMDP exact solution algorithm, nor the first one to use its core principle of generating and pruning vectors, yet it is very straightforward and it can be considered a "common denominator" of some more advanced algorithms.

The Exhaustive Enumeration algorithm follows the value iteration dynamic programming principle as shown for MDPs in section 2.2.1. It starts with setting the zero-length horizon value function to be always zero, using a single zero vector to represent it:

$$\forall b \in \Delta(\mathcal{S}) : V_0^*(b) = 0 \qquad\qquad \mathcal{V}_0 = \{(0)_{|\mathcal{S}|}\}$$

The value iteration step follows the Bellman optimality equation for POMDPs (eq. 3.2). Since the previous step value function is expressed as a (maximum over products of a) set of vectors, we get

$$\begin{aligned} V_{k+1}^*(b) &= \max_{a\in\mathcal{A}} \left\{ \rho(b,a) + \gamma \sum_{o\in\mathcal{O}} z(b,a,o) V_k^*\big(BU(b,a,o)\big) \right\} \\ &= \max_{a\in\mathcal{A}} \left\{ \rho(b,a) + \gamma \sum_{o\in\mathcal{O}} z(b,a,o) \max_{\alpha\in\mathcal{V}_k} \alpha BU(b,a,o) \right\} \\ &= \max_{\substack{a\in\mathcal{A} \\ \forall o\in\mathcal{O}:\, \alpha_o\in\mathcal{V}_k}} \left\{ \rho(b,a) + \gamma \sum_{o\in\mathcal{O}} z(b,a,o) \big(\alpha_o BU(b,a,o)\big) \right\} \\ &= \max_{\substack{a\in\mathcal{A} \\ \forall o\in\mathcal{O}:\, \alpha_o\in\mathcal{V}_k}} \left\{ \rho(b,a) + \gamma \sum_{o\in\mathcal{O}} \alpha_o O_o T_a b \right\} \end{aligned} \tag{3.5}$$

Using eq. 3.4 backwards, we can express this equation as a vector operation

$$\mathcal{V}'_{k+1} = \left\{ \rho_a + \gamma \sum_{o\in\mathcal{O}} \alpha_o O_o T_a \;\middle|\; \begin{array}{l} a \in \mathcal{A}, \\ \forall o \in \mathcal{O} : \alpha_o \in \mathcal{V}_k \end{array} \right\} \tag{3.6}$$

where ρ_a is the reward vector for each action ($\rho_a(s) \stackrel{\text{def}}{=} R(s,a)$).

Note that this procedure exponentially increases the number of vectors in the vector set: for each action we have to choose all possible $|\mathcal{O}|$-tuples (one for each observation) of vectors from the previous step value function set, i.e. $|\mathcal{V}'_{k+1}| = |\mathcal{A}||\mathcal{V}_k|^{|\mathcal{O}|}$.

It turns out that many of these vectors are dominated by the remaining vectors ($\alpha \in \mathcal{V}$ is dominated when $\forall b \in \Delta(\mathcal{S}) : \exists \beta \in \mathcal{V} : \beta b \geq \alpha b$) and

can therefore be removed from the set without affecting the value function according to eq. 3.4. The process of finding all dominated vectors in the set and removing them is called *pruning*[2] and it completes the value iteration step in the Exhaustive Enumeration algorithm:

$$\mathcal{V}_{k+1} = \text{prune}(\mathcal{V}'_{k+1})$$

It is realized by solving a linear programming problem for each vector $\alpha \in \mathcal{V}'$:

$$\max_b \{x : x \leq (\alpha - \beta)b, \alpha \neq \beta \in \mathcal{V}', x \in \mathbb{R}\}$$

If the solution is greater than zero, the vector α is not dominated and therefore it cannot be removed from the value function vector set.

The stopping condition remains analogous to that in MDP value iteration algorithm:

$$\forall b \in \Delta\mathcal{S} : |V_k(b) - V_{k-1}(b)| < \epsilon$$

When this becomes true, the current value function V_k^* is declared final.

Representing the optimal policy

Although the optimal policy is entirely determined by the value function and being able to evaluate the value function allows the agent to find the action with the highest expected reward, such a reasoning would be unnecessarily complex. For each possible action, the agent would have to consider all possible observations that it could receive and evaluate the value function (i.e. maximize a set of size $|\mathcal{V}|$) at the belief state that such an observation would induce; this gives a complexity of $\mathrm{O}(|\mathcal{A}||\mathcal{O}||\mathcal{V}|)$ for each time step.

Note that at eq. 3.6, each vector is associated with a choice of action for the next time step and when evaluating the value function at a specific belief state using eq. 3.4, the maximization corresponds to, among others, the choice of best next action (see eq. 3.5). Therefore, if we mark each vector in the set with the next action it corresponds to and select the action corresponding to the maximum vector product in eq. 3.4, finding the optimal action is simplified to $\mathrm{O}(|\mathcal{V}|)$.

Though it may seem hard to believe, the reasoning can be simplified even further to a constant time ($\mathrm{O}(1)$). For many problems, all the belief states that produce maximum with one vector will be transformed via belief update to belief states also sharing a single maximizing vector, given

[2] In some articles it is called *purging*.

the single optimal action and a single resulting observation. This allows as to construct a simple *finite state machine* of size $|\mathcal{V}|$, sometimes called a *policy graph*, where nodes correspond to portions of belief space sharing the same maximizing vector, each node has an associated optimum action, transitions are given by the belief update process having observations as input and the initial node is the portion of belief space that b_0 belongs to. Having and following this policy graph, the agent doesn't need the knowledge of belief state anymore, nor has it to calculate the belief update for optimal reasoning. For problems where size of the (sub)optimal policy is significantly lower than number of states, this means a great reduction of space complexity in addition to improving time complexity.

3.2.2 Incremental Pruning

The Exhaustive Enumeration algorithm, being simple to explain, has a very bad time complexity. In the pruning step, for each vector in the exponentially blown up vector set we have to solve a linear programming problem of the size of the vector set. In fact, this proves intractable for all but very small problems [Cassandra et al., 1997].

The Incremental Pruning algorithm [Cassandra et al., 1997] improves the Exhaustive Enumeration using a simple but fundamental thought. Let us define the *cross-sum* set operator $\oplus$ as sums of all possible combinations of elements in the sets:

$$\bigoplus_{i=1}^{n} A_i = A_1 \oplus A_2 \oplus \cdots \oplus A_n \stackrel{\text{def}}{=} \left\{ \sum_{i=1}^{n} x_i \,\middle|\, \forall i = 1 \ldots n : x_i \in A_i \right\}$$

Using this notation, we can rewrite the value iteration step of the Exhaustive Enumeration algorithm as

$$\begin{aligned} \mathcal{V}_k &= \text{prune}\left(\bigcup_{a \in \mathcal{A}} \mathcal{V}_k^a \right) \\ \mathcal{V}_k^a &= \bigoplus_{o \in \mathcal{O}} \mathcal{V}_k^{a,o} \\ \mathcal{V}_k^{a,o} &= \left\{ \frac{\rho_a}{|\mathcal{O}|} + \gamma \alpha O_o T_a \,\middle|\, \alpha \in \mathcal{V}_{k-1} \right\} \end{aligned}$$

Then the fundamental thought of incremental pruning is the fact that

$$\text{prune}(A \oplus B \oplus C) = \text{prune}\left(\text{prune}(A \oplus B) \oplus C \right)$$

and also

$$\text{prune}(A \oplus B) = \text{prune}\left(\text{prune}(A) \oplus \text{prune}(B)\right)$$

Using this, we get an equivalent algorithm by pruning the set after adding every observation to the cross-sum operation as well as after the vector set transformation:

$$\mathcal{V}_k = \text{prune}\left(\bigcup_{a \in \mathcal{A}} \mathcal{V}_k^a\right) \tag{3.7}$$

$$\mathcal{V}_k^a = \text{prune}\left(\ldots \text{prune}\left(\text{prune}(\mathcal{V}_k^{a,1} \oplus \mathcal{V}_k^{a,2}) \oplus \mathcal{V}_k^{a,3}\right) \oplus \ldots\right) \tag{3.8}$$

$$\mathcal{V}_k^{a,o} = \text{prune}\left(\left\{ \frac{\rho_a}{|\mathcal{O}|} + \gamma \alpha O_o T_a \,\middle|\, \alpha \in \mathcal{V}_{k-1} \right\}\right) \tag{3.9}$$

The authors of this algorithm have shown that the number of linear programming problems for each value iteration step is only linear to the number of observations instead of being exponential. This, in combination with smaller size of the linear programming problems, makes Incremental Pruning one of the fastest generic POMDP exact solution algorithms nowadays [Cassandra, 1998].

Part II

Solution of a model problem

Chapter 4

Description of the problem

I will demonstrate the potential of POMDP framework for dynamic troubleshooting on a problem derived from a well-known synchronization problem of *dining philosophers*.

4.1 Motivation

Like with the original synchronization problem, the dining philosophers may be considered a metaphor for a large system composed of partially independent processes, some of which share certain resources that cannot be used by more than one process simultaneously. The original problem dealt with the potential issue of multiple processes waiting in a deadlock for one or more resources that cannot be freed, while the POMDP problem "Dining Philosophers with Falling Sticks" targets the dynamic troubleshooting problem of identifying the cause of error and reasoning about the costs and benefits of rectifying it. The deadlock issue is left to be solved by known means, instead the interconnection of the processes via the shared resources is used as a challenge for the generality of the POMDP framework.

4.2 Problem definition

There are n philosophers sitting at a round table, each one of them having a plate with rice in front of him. Initially, all of the philosophers are (deeply) thinking and there are n forks lying on the table, one between each two neighbouring plates.

A philosopher might get hungry at any time and if he wants to eat, he needs to use both of the forks adjacent to his plate. If both of them

are available, he starts to eat. An eating philosopher may similarly get satiated at any time and stop eating immediately. However, there is a chance that instead of returning both of the forks he was using on the table, he might drop one or both of them on the floor without realising it. When a philosopher gets hungry and he cannot find one or both of his forks, he doesn't care whether it is caused by his neighbour(s) eating or a fork (forks) having been dropped, he simply signals the waiter that there is a problem by ringing and then returns to his thinking.

The agent in this problem is represented by the waiter. He can hear the philosophers ringing and distinguish which one of the philosophers has been ringing. However, he doesn't see the table, therefore he cannot tell which forks are being used and which have been dropped. Whenever he decides to do so, he can walk to the table and replace a single fork in case it has been dropped. If he finds out that that fork is being used or is lying on the table, he has wasted a walk. As the waiter is lazy and gets annoyed easily, his goal is to minimize both visits to the table and the amount of times some philosopher(s) ring(s).

4.2.1 POMDP formalization

The state of the environment is given by the state of all the philosophers (thinking or eating) and the state of all the forks (lying available on the table, being used by a philosopher or having been dropped on the floor). Therefore, the upper bound for the number of states is $2^n 3^n = 6^n$. Of course not all of these combinations are valid states and the true number of states is somewhat smaller.

At any (discrete) time step, any of the philosophers can be ringing. That gives us 2^n observations. However, at each time step the waiter can be checking the state of at most one of the forks. He manages to do a single check – walk to the table, pick up a fork if it has been dropped and walk back – in a single time step, that makes $n + 1$ possible actions (n forks and doing nothing).

The process follows the Markov assumption and the philosophers and forks are symmetrical, so for a complete transition and observation model we have to determine these probabilities:

- $p_{hungry} + p_{think\_on} = 1$: probabilities for a single thinking philosopher to get hungry vs. keep thinking,
- $p_{satiated} + p_{eat\_on} = 1$: probabilities for a single eating philosopher to get satiated vs. keep eating and

- $p_{drop\_fork} + p_{lay\_fork} = 1$: probabilities for a single fork being laid down on the table to fall on the floor or not.

For the agent's reasoning we need the reward model specified by

- r_{cannot_eat}: the negative reward (i.e. cost) for one hungry philosopher not being able to eat and ringing,
- r_{pick_up}: the negative reward for walking to the table, no matter whether the fork gets replaced or it doesn't need to, and
- γ: the reward discount factor at each time step.

Putting these sub-models together as conditionally independent where possible, we will encounter the original synchronization problem: consider two neighbouring philosophers, both of them are thinking, all the 3 adjacent forks are lying on the table and both philosophers get hungry in a single time step. Which one gets the middle fork and which one has to ring?

Perhaps the simplest solution to this from the implementational point of view is to evaluate the philosophers sequentially. We will divide each time step to $n+1$ micro-steps for each atomic state change. At micro-step 0 the waiter comes if he has decided to and checks one of the forks. Then, at micro-step 1 the first philosopher is allowed to start or stop eating (while picking up, laying down or dropping forks), at micro-step 2 the second philosopher gets this opportunity and so on until the last philosopher. After that the total reward for this step is deducted, the waiter receives his observations, decides his next action and the whole process repeats in a next regular time-step. This described procedure resolves all ambiguities and can be easily translated to the regular whole-process transition model.

One important property of this approach is that it breaks the symmetry of the philosophers and the forks. If two neighbouring philosophers want to start eating, the one that is assigned an earlier micro-step gets the fork, therefore his probability to start eating is different from the other one's. Considering random assignments of micro-steps for each regular time-step and translating that to the whole-process transition model would bring the symmetry back, yet for our purpose it is not necessary and the asymmetry of the model can even be considered somewhat beneficial.

4.3 Solutions used till now

In his Bachelor's thesis, Svatopluk Ledl [Ledl, 2004] implements a program for simulating the Dining Philosophers with Falling Sticks problem

outside the POMDP framework. In his program, he includes a "Minimal Intelligence Serviceman" (i.e. waiter) algorithm that offers simple three-parameter solution of this problem, and a learning algorithm for these parameters. No other solution attempts for such a problem are known to the author at this time.

Chapter 5

The proposed heuristic

In this chapter I will present a heuristic that is the main contribution of this work. Although it was designed to solve the model problem and on account of that I will concentrate on this specific application while describing it, it contains some generic principles that could be easily adapted to a wider class of problems.

5.1 Motivation

The model problem, when solved exactly, suffers from great time and space complexity which allows only very small versions to be tractable. Given n philosophers, size of the state space is $|\mathcal{S}| \in \mathrm{O}(2^n)$ and the observation space size $|\mathcal{O}|$ is exactly 2^n. After each value iteration step, the number of vectors grows up to exponentially in $|\mathcal{O}|$, that is up to superexponentially in n. These vectors have to be stored in memory, so this superexponential complexity applies to space as well as time.

On the other hand, it is clear that the more philosophers we have, the weaker dependence there is between the state of more distant, say opposite, philosophers and forks. From the definition of the problem it is clear that for a single time step, state of a certain philosopher is conditionally independent on any other state variable given the state of the adjacent forks. As we increase the time horizon, the space variable dependency increases: for a given event (state change of a single state variable, e.g. a philosopher drops a fork), the range of other state variables that might get affected by this event grows linearly with time until all of the variables might get affected. The more philosophers there are, the longer time horizon it takes for a single event to potentially influence all of the state variables.

Like many other more or less generic heuristics, this heuristic uses factorization of the state space to take advantage of the weak dependencies between certain state variables and to reduce the size of both state and observation space. The variables of the state space are divided (not disjunctively) into weakly dependent *subprocesses* that are considered independent for the purpose of maintaining a belief state. The modified belief update process provides exchange of information between the subprocesses, yet it never needs to manage the whole original state space.

The accuracy of such an approach has been studied intensively in [Boyen and Koller, 1998]. It has been shown that for many problems, any error brought by this approach doesn't increase with time; on the contrary it decreases exponentially and the overall error remains bounded.

5.2 Process factorization

In the original problem, there are two types of state variables – philosophers and forks, and one type of observation variable – ringing from a philosopher. A *subprocess* is a small collection of state and observation variables. To bring any benefit, these variables should be more strongly dependent with each other than with other state and observation variables.

In this algorithm, a subprocess is a collection of adjacent philosopher and fork variables with odd number of forks and even number of philosophers (i.e. fork–centred) and the observations (ringings) corresponding to those philosophers. An example would be a collection "P—F—P" or "F—P—F—P—F" where F stands for a fork and P for a philosopher. Such a subprocess has a fixed size of state space ($\bar{\mathcal{S}}$) and observation space ($\bar{\mathcal{O}}$).

5.2.1 Belief state representation

To describe the whole process, we have n similar subprocesses – each fork is a central fork in one of the subprocesses. Although the total number of state and observation variables in all of the subprocesses is greater than in the original problem, we consider these subprocesses independent for purposes of storing a belief state.

The belief state in this algorithm is a not a single probability distribution over the states of the whole process, but a collection of probability distributions – for each subprocess a distribution over possible states of the subprocess. The belief state space is therefore $\Delta(\bar{\mathcal{S}})^n$ and the amount

of memory for storing a single belief state is $n|\bar{\mathcal{S}}|$, in contrast to $\mathrm{O}(2^n)$ for the exact solution.

5.3 The belief update process

The belief update is the most complex process involved in this heuristic. It has to be run for each subprocess without reasoning about the whole process (we expect the whole state space to be unrepresentably big), yet it should allow the modelled subprocesses that are more strongly dependent to interchange some information. Suppose we are calculating the belief update for one of the subprocesses, I will refer to it as *core subprocess*.

To achieve interchange of information, most of the belief update process is run not on the core subprocess itself, but rather on a *greater subprocess* of it. By that we mean the union of state and observation variables from the core subprocess and from some or all (by specification) other subprocesses that share at least one state variable with the core subprocess, these subprocesses will be referred to as *connected subprocesses*. In the simplest example where a subprocess is a "P—F—P" collection, we could add state variables from the two adjacent subprocesses to form a "P—F—P—F—P—F—P" greater subprocess.

The belief update process of a single (core) subprocess is composed of three phases:

1. Collect information from belief states of the connected subprocesses to calculate the belief state for the greater subprocess.
2. Perform a belief update on the greater subprocess.
3. Extract the information from the greater subprocess's new belief state to get the new belief state of the core subprocess.

5.3.1 The combining phase

By combining two subprocesses we mean taking their belief states and calculating a combined belief state – probability distribution over all possible value assignments to the state variables from those subprocesses. For clearer explanation, I will present a sequential combining process that combines only two subprocesses at once, though an equivalent but faster single-phase computation is implemented further in this work.

The combining sequence always starts with the belief state of the core subprocess, adding connected subprocesses one-by-one by forming a belief

state for the union of the state variables of the combined subprocesses. In this sequence, the next subprocess to be combined should always be the subprocess that shares most state variables with the already combined set, i.e. in the dining philosophers problem that would be one of the adjacent subprocesses.

Let us denote subprocess 1 the core subprocess (or the currently combined set) and subprocess 2 the subprocess that should be combined with it. We have three disjoint sets of state variables that together form the resultant subprocess combination: variables in subprocess 1 but not in subprocess 2 are marked A, variables in both subprocesses are marked B and variables only in subprocess 2 are marked C. The belief state of subprocess 1 gives us probability distribution over combinations of assignments to variables from $A \cup B$, belief state of subprocess 2 for variables from $B \cup C$ and we need to calculate probability distribution over complete assignments to $A \cup B \cup C$. Let us denote these assumed probabilities P_1, P_2 and P_{12} respectively. Suppose we are calculating the probability of one specific assignment abc.

The main idea is to use probability assignments from subprocess 1 as basic and from subprocess 2 to use only conditional probabilities for extending the probability distribution with the new variables:

$$\begin{aligned} P_{12}(abc) &= P_1(ab)P_2(c|ab) = P_1(ab)P_2(c|b) = P_1(ab)\frac{P_2(bc)}{P_2(b)} = \\ &= P_1(ab)\frac{P_2(bc)}{\sum_i P_2(bc_i)} \end{aligned} \tag{5.1}$$

where $(c_i)_i$ are all possible assignments to variables in C. Repeated for every possible assignment to $A \cup B \cup C$ (i.e. for every state of the combined subprocess), we get the complete belief state. Repeating this whole process for each connected subprocess, we get the belief state for the greater subprocess.

5.3.2 The greater subprocess belief update

This is basically a belief update for a modified Dining Philosophers POMDP where instead of round table there is a straight (one–side) table and the philosophers and their forks form a line. This is an outline of the main differences from the original POMDP:

- If at the ends of the table there are philosophers (not forks – like in the "P—F—P—F—P—F—P" example), then a philosopher at the end needs only one fork to eat.

- The negative reward for checking the state of a fork is deducted only if it is the central fork of the core subprocess.
- The negative reward for the ringing philosophers that are not in the core subprocess is ignored, for philosophers in the core subprocess it is divided by the amount of them.

From the received observation in the whole process only the variables in this greater subprocess are extracted and such an observation is used for the belief update. The action for the whole process is transformed likewise (if the whole–process action is picking up a fork that is not in this greater subprocess, then the action for this subprocess is "do nothing"). After these transformations, the belief update of the just–described greater subprocess is calculated.

5.3.3 The extraction phase

Following the nomenclature from 5.3.1, let A be the set of variables in the core subprocess and B the set of all other variables in the greater subprocess. Probabilities inferred from the core subprocess belief state are marked P_1 and probabilities from the greater subprocess belief state are marked P_{12}. While calculating the probability of a single assignment (subprocess state) a, we have

$$P_1(a) = \sum_i P_{12}(ab_i) \tag{5.2}$$

where $(b_i)_i$ are all possible assignments to variables from B.

5.3.4 Estimating the likelihood of observations

Strongly related to calculating the belief update for a given action and observation is estimating the likelihood of that observation. In a traditional POMDP belief update, this value is available instantly as it is the normalizing constant used in the belief update function (see eq. 3.1). Being able to compute both the belief update and to estimate the likelihood of observations, one can calculate the value iteration step and therefore solve (at least point-wise) the whole POMDP problem.

Unfortunately, estimating likelihoods of observations is not so straightforward while using the heuristic. Although we get a likelihood of observation for each greater subprocess while performing its belief update, these likelihoods can vary greatly. For example, suppose that in our belief state

a certain philosopher has currently a high probability of eating. If the waiter receives a ringing from that philosopher, such an observation would have a small likelihood, because an eating philosopher doesn't ring. Using the heuristic, even if the belief states for the subprocesses were perfectly consistent, this "contradiction" would have no effect on subprocesses not containing that philosopher and these subprocesses could therefore produce much higher observation likelihood.

As there is no global information about the belief state, the resultant whole–process likelihood has to be obtained from the likelihoods from subprocesses. As with the belief update itself, we will assume independence of the subprocesses. If we had only one philosopher per subprocess, that would mean also independence of the observations and we could simply use a product of the likelihoods. Because each philosopher is present in m subprocesses when there are m philosophers per subprocess, the likelihood estimation formula for this heuristic is given as

$$P(o|b) = \left(\prod_i P_i(o_i|b_i) \right)^{\frac{1}{m}} \tag{5.3}$$

where $P_i(o_i|b_i)$ is the likelihood of observation o under belief state b as computed in 5.3.2 for the subprocess i.

In case of less consistent belief states for the individual subprocesses, these estimated likelihoods for all of the possible observations are not guaranteed to have a sum of 1. Since in the value iteration step these likelihoods are used to weight the expected rewards of possible belief state evolutions, they need to be computed completely in advance and normalised before being used as individual weights.

5.4 Modification of the heuristic for vector representation

Being able to represent a belief state and perform a belief update are the necessities to use a model in simulation. However, for some simple solution algorithms that use only finite number of points to represent a value function (which is then required to be continuous), this is also enough to run a value iteration and compute the value function, getting hopefully near-optimal solution of the problem as a whole.

Unfortunately, this doesn't apply to the exact algorithms like incremental pruning. These algorithms require the value function to be PWLC. In the regular POMDP model, this is guaranteed by the fact that the

value iteration step, when expressed algebraically, is actually a maximum from finitely many linear transformations of the original value function. In case of the presented heuristic, this is not true: phase one of the belief update uses multiplication and division of some values from the prevous belief state (see eq. 5.1). As the belief update isn't linear anymore, neither is the value iteration.

In this section I present a modification of the heuristic that keeps the belief update to be a linear transformation. As a result of that, such a heuristic can be incorporated in a vector-based algorithm to exploit its advantages.

What needs to be modified is the first phase of the belief update. It has to be made a linear transformation, therefore only addition, subtraction and multiplication of the original values by a coefficient may be used. The input of this phase is two or more probability distributions – belief states for the subprocesses, the output is a probability distribution for a greater subprocess.

Considering the factual aspect of this computation, for every state of the greater subprocess we are inferring its probability given some (also inferred) probabilities of the subprocesses. The subprocess probabilities could be considered marginals of the greater subprocess probabilities; on the other hand, the actual values of the probabilities of different subprocesses are not guaranteed to be consistent because of the approximation nature of the algorithm.

The solution proposed here is for each state of the greater subprocess to take a weighted sum of the corresponding probabilities of the subprocesses. Following the nomenclature from 5.3.1, this is

$$P_{12}(abc) = \omega_1 P_1(ab) + \omega_2 P_2(bc)$$

By setting weights 1 for the core subprocess and 0 for the connected subprocesses, we are ignoring any information from the connected subprocesses and we are expecting uniform probability distributions for the respective state variables given the variables of the core state. Any other (non-negative) weight combination has no theoretical basis, yet in fact the resultant probabilities will lie between the probabilities appointed by the specific subprocesses, which is what we originally wanted to accomplish. The actual weights have to be chosen by trial and error. Weight combinations that should be considered are equal weights as well as weights favouring the core subprocess or subprocesses that have more common variables with the core subprocess.

For any weight combination, the resultant belief state has to be normalized, which can be accomplished by normalizing columns of the

transformation matrix and dividing the whole matrix by the number of subprocesses being combined.

Chapter 6

Proof-of-concept implementation

Four complete implementations of the problem solution altogether are a part of this work. Originally aimed as a proof-of-concept implementations, they were programmed in Python which, being and interpreted language, is not an ideal tool for intensive computations. However, most of the computationally intensive procedures were implemented as vector and matrix operations that are performed within the compiled *NumPy*[1] library and some of these procedures were parallelised to exploit full computational power of multiprocessor machines. Also, the structure of the algorithms was made as generic as possible, allowing the code to be adapted for possible future implementations of other POMDP problems. In this chapter I will provide a brief explanation on used algorithms and techniques. More detailed description may be obtained from the inline comments of the code or by contacting author.

Two of the implementations use the proposed heuristic. The first one uses the original heuristic and is implemented via fixed grid approximation of the value function while the other one uses the modified vector version of the heuristic and the incremental pruning algorithm. For a comparison of results, two generic solution algorithms have been implemented: the fixed grid approximate algorithm and the incremental pruning algorithm. All of the code was programmed from scratch entirely for purpose of this work, except for the vector set pruning phase of the incremental pruning algorithm that has been taken from Anthony R. Cassandra's "pomdp-solve" program[2] [Cassandra, 2005]. That and the interprocess

[1] Although not officially part of Python, it is the de-facto standard library for array, vector and matrix computations.

[2] Released under the GNU General Public License.

communication interface is the only part of the program written in C, everything else is written in Python.

6.1 The Dining Philosophers POMDP

The problem, as defined in section 4.2.1, is too complex to be specified by human as rigorously as a required by the POMDP framework definition (section 3.1). To overcome that, a Python code has been programmed that takes as an input only the parameters specified in section 4.2.1 and produces both auxiliary functions for enumerating states, actions and observations as well as POMDP data required by generic algorithms – the transition model, the reward model and the state estimator. In fact, this code, shared among all of the further described implementations, has been made generic enough to cover also the linear table configurations of the (greater) subprocesses required by the heuristic. This code is placed in source files `gen_pomdp.py` and `gen_se.py`.

Class `State` is used to represent a single state of a POMDP and it provides easy transformations of the state that might happen during the process. This class also contains tools to generate and enumerate all possible states.

Class `GenericPomdp` implements the static part of the dining philosophers POMDP. When creating an instance of `GenericPomdp`, all possible states are generated according to specified parameters and they are enumerated (allowing for fast transition between the state number and the `State` instance). Basic tools for enumerating and manipulating actions and observations are also included.

Class `GenericStateEstimator` contains all of the code simulating the dynamics of the dining philosophers POMDP. When an instance of it is initialised, a state transition model is generated using the sequential philosopher evaluation approach from section 4.2.1. After that, the state estimator matrices for each action and received observation are generated along with the reward vectors for each action. The `GenericStateEstimator` class code contains also various tools for manipulating the belief state and for running a simulation of the process.

The state estimator matrices are the main difference from the formalism stated in chapter 3, where we had separate (diagonal) observation matrices for each observation and transition matrices for each action and we multiplied them when necessary. Though such an arrangement would require less memory, it cannot be used in the formalism that we have been using for our model problem so far, that is where observations are made

dependent not only on previous state but also on the previous action[3].

6.2 The original heuristic algorithm

As the original heuristic doesn't guarantee to have a PWLC value function and therefore it cannot be represented as a maximum over a set of vectors, a uniform-spaced fixed grid approximation of the value function has been implemented. As a comparison, a generic value iteration algorithm using similar approximation of the value function has been programmed as well.

6.2.1 Fixed grid approximation

The approximation of the value function uses a very simple principle: the belief space is divided using a fixed, uniform-spaced grid. At each value iteration step, the value function is computed only at the points of the grid. When evaluating the value function at a given point (belief state), linear interpolation of the nearest grid points is used.

Though the outlined principle is simple, the implementation is not trivial. As the belief space is not $\mathbb{R}^n$ but rather the simplex $\Delta(n)$, these issues arise:

- How to efficiently generate all of the grid points of the simplex $\Delta(n)$?
- How to represent the gridded belief space considering that using a portion of $\mathbb{R}^n$ would be a significant waste of space for high n's?
- Given an arbitrary belief state (i.e. point in the simplex), how to identify the nearest neighbour grid points and how to interpolate the values at them?

The code responsible for these tasks resides in the source file `fixgrid.py` in the class `SimplexArray`. Instances of this class represent grid points of an n-dimensional simplex with m grid points per dimension and values assigned at each point. As computing some algebraic isomorphism between the grid point coordinates and a continuous range of natural numbers would not be simple, the hash table implementation available in

[3]Should memory requirements become an issue, a simple change in the definition of the sequential philosopher evaluation – making the waiter check and possibly pick up the forks *after* all of the philosophers' state changes have been evaluated – would allow the use of the formalism from chapter 3.

Python is used instead. When an instance of `SimplexArray` is initiated, this bidirectional mapping (if not found in cache) is constructed. All of the grid point coordinates are generated sequentially in a lexicographic order omitting the last dimension, these coordinates are stored in an array and a hashtable mapping from the coordinates to naturals is filled in.

The last but hardest task, identifying the grid points that form a grid-spacing size simplex that contains an arbitrary simplex point and calculating distances to them, is implemented using the *Freudenthal triangulation* explained in [Lovejoy, 1991a]. It is an iterative process that can be realised only in algebraically different coordinate system called Freudenthal coordinates. Since the probability distribution in terms of simplex algebra means using the traditional barycentric coordinates, a back and forth coordinate transformation has to be performed upon every interpolation. For a description of these calculations, please see [Lovejoy, 1991a].

6.2.2 Value iteration algorithm

The core of the value iteration algorithm implementation is programmed in the source file `fixvaliter.py` and it is used by both the generic algorithm and the heuristic algorithm.

The most important part of it is the function that performs a single value iteration step. It uses the value function array object (`FixedGridValueFunction` or `GridHeuristicValueFunction`) object to walk through the grid points of the simplex. At each point (a certain belief state) and for every possible action, it uses either the `GenericStateEstimator` or the `GridHeuristicSE` object to generate all possible transformations of the belief state through a received observation and an estimated likelihood of such a transformation. Using the value function object it evaluates the current value function at transformed belief states and by weighting with the estimated likelihoods it computes an expected reward for each action at every grid point. Selecting the most worth action at each grid point it gets values of the new value function and stores these them in a new value function object.

This code was completely parallelised using Python's `multiprocessing` module. At first, a pool of worker processes (using number of CPUs provided by the operating system) is forked and every one of the worker processes uses shared memory to obtain its own copy of the current value function object and the state estimator object. Then the worker processes are assigned short sequences of grid points and when they compute values of the new value function at them, they return them to the master process

that stores them in the new value function object. Assignment of the grid points to the worker processes is performed dynamically whenever a worker process has finished its current job.

6.2.3 The heuristic itself

Implementation of the original heuristic can be found in the source file `gheur_se.py` within the class `GridHeuristicSE`. In this implementation, the greater subprocess is formed by the combination of the core subprocess and two adjacent subprocesses, regardless of their size (see section 5.3).

Value function

Following the definition of the belief state in the heuristic (section 5.2.1), the belief state space is $\Delta(\bar{\mathcal{S}})^n$ where n is the number of subprocesses and this determines the representation of the value function. It is an array of n dimensions and the dimension indexes represent the grid points of the respective simplexes, i.e. belief states of the respective subprocesses. A single value of that array therefore corresponds to a combination of per-subprocess belief states. A code responsible for creating and manipulating this representation of the value function is in the class `GridHeuristic-ValueFunction`. The mapping between simplex grid points and a range of natural numbers from the `SimplexArray` class is used to index the dimensions of the array.

Although this representation of the value function reduces the required space from superexponential to exponential in the number of states, the time complexity of retrieving a single interpolated value from the value function would be unusably high if used rigorously: one would need to find the grid points of the smallest bounding simplex for each subprocess and then to retrieve every possible combination of one of these points per subprocess. For example, if the dimensionality of the simplex (number of states in a subprocess) were d, then d^n points would need to be retrieved from the array to get just one interpolated value.

Given that a distance metric in this belief space is a product of distances per each simplex, the current implementation uses a heap-based algorithm[4] that finds only $2n$ nearest grid points and returns an inverse–distance–weighted average of them.

[4]The algorithm actually works with logarithms of the probabilities to replace products by sums.

Initialization

Many computations are concentrated in the initialization code for `GridHeuristicSE` instances, so that as least as possible computations are left for the functions used later in loops. At first, three `GenericPomdp` objects of the type with linear table are created: one of them represents a subproces, one represents a greater subprocess and one represents the intersection of variables of two adjacent subprocesses. These all would be used for enumeration of states and state transformations of the subprocesses. One `GenericStateEstimator` object that will serve as a state estimator for the greater subprocess is initialized as well.

Then the preparation for each phase of the belief update is done. Phases two and three (greater state belief update and extraction phase) are both linear transformations; for the former we already have the state estimator matrices (one for each combinantion of action and observation) generated by the `GenericStateEstimator` object and for the latter a matrix that performs the sum from eq. 5.2 is generated easily using state enumerations and manipulations provided by the `GenericPomdp` and `State` objects. These matrices are then multiplied to save computation, because within the belief update the belief state would be multiplied by them sequentially.

Phase one – subprocess belief state combination – is not a linear transformation and that is the reason why value function approximation has to be used at all. This operation consists of sums, multiplications and divisions of probabilities (see eq. 5.1) and therefore it cannot be prepared ahead of time in the form of a matrix. Yet what can be prepared are the indices of probabilities that are to be summed up, multiplied and divided; this is the last thing performed in the initialization of `GridHeuristicSE` objects.

Belief update

Thanks to these preparations, the belief update provided by this class is made relatively simple. As stated in section 5.3, it has to be performed for each subprocess.

Since we are combining exactly three subprocesses into a greater subprocess, we don't need to combine them sequentially like in section 5.3.1 and we can calculate the combination in a single equation. Let a, b, c, d and e be an assignment to state variables:

- a to variables from the left-hand connected subprocess only,

- b to variables from the intersection of the left-hand connected subprocess with the core subprocess
- c to variables from the core subprocess except the two mentioned intersections,
- d to variables from the intersection of the core subprocess with the right-hand connected subprocess and
- e to variables from the right-hand connected subprocess only

and let the probabilities inferred from the left-hand subprocess belief state, core subprocess belief state, right-hand subprocess belief state and greater subprocess belief state be marked P_1, P_2, P_3 and P_{123} respectively. Then the combination equation is

$$P_{123}(abcde) = \frac{P_1(ab)}{\sum_i P_1(a_i b)} P_2(bcd) \frac{P_3(de)}{\sum_j P_3(de_j)}$$

where a_i and e_j are all possible assignments to the respective variables.

At first the probabilities for the intersection subprocesses are summed up and the results are cached for they will be used more than once. Then for every assignment of *abcde* (i.e. greater subprocess state), the prepared indexes are used to select the three probabilities from the belief states and two sums from the prepared sums which corresponds to the five factors of the equation. Finally these five numbers are multiplied / divided according to the equation. Like in most of the code in this work, the selection and the multiplication / division is realised as a NumPy array operation to avoid loops that are very slow in interpreted languages like Python.

The implementation takes advantage of the typical sequence of operations in a value iteration step. It allows to use an auxiliary `GridHeuristicPreparedBS` object that represents a performed phase one of the belief update for all of the subprocesses (phase one depends only on belief state, not on action or observation). It can then be reused many times for phases two and three that are iterated for every action and observation. The latter realisation of phases two and three is much simpler and consist of multiplication by the prepared matrix. Then the resultant belief states are normalized according to eq. 3.1 and combined into belief state of the whole process. The estimation of observation likelihood according to eq. 5.3 is computed as well.

6.2.4 Testing code for simulations

A generic POMDP simulation program is implemented in the source file `pomdp_test.py`. It is used to simulate policies from all four problem solving implementations. As an input it requires the resultant value function from one of the implementations, a state estimator used by that implementation for performing belief updates and a true transition model that is always provided by the class `GenericStateEstimator`. It then performs a requested amount of fixed length simulations and returns the final discounted reward.

Since solving the problem usually takes a lot more time than running the simulations, only the generic simulation algorithm that chooses the best action by evaluating all possible belief state transformations is implemented. For the vector–based algorithm, the simulation could be done faster by having stored the most worth actions for each portion of belief space as described in section 3.2.1, but there is no such improvement for the algorithms that use grid approximation of value function. The testing code is completely parallelised in a way described in section 6.2.2.

6.3 Vector enabled heuristic

A modified version of the heuristic with linear belief update has been implemented and incorporated to the incremental pruning algorithm, as was suggested in section 5.4. Along with that, a generic version of the incremental pruning algorithm was programmed for comparison. Both of these implementations take advantage of the existing "pomdp-solve" program's routine for pruning vector sets.

To overcome issues connected with using some compiled shared libraries in Python, the `main()` function of the pomdp-solve program has been rewritten to ignore most of the program code and to serve as a vector pruning service using interprocess communication implemented via pipes. During execution of the Python solution algorithms, proportion of the computing time that is spent within the pomdp-solve process is measured. The realised measurements of these times indicate that for all but the simplest configurations, an overwhelming majority of the execution time is spent within the compiled C code of the vector pruning routine, therefore using Python for the rest of the program practically doesn't affect the performance at all.

6.3.1 Incremental pruning

Code for a single value iteration step of the incremental pruning algorithm resides in the source file `incprune.py` within the class `IncPruner`.

The `iterate_vf()` method is responsible for calculating the equation 3.7, e.g. union of the available actions and the final vector pruning. For every action it calls the method `_recalc_vf_action()` that realizes the equation 3.8 by incrementally adding every possible observation, pruning the current set after every cross-sum operation. In addition to that, a reward vector of the current action is added to the set of the first observation. This is the only difference from the equations in section 3.2.2, where a fraction of this vector is added to every transformed vector in eq. 3.9.

The rest of equation 3.9 is computed via the method `_project_vf()` that is run for every action and observation. Multiplication by the discount factor γ has been done ahead of time in this class's initialisation code by multiplying the whole state estimator matrix with this factor.

6.3.2 The modified heuristic

A complete implementation of the modified heuristic from section 5.4 can be found in the source file `vheur_se.py` within the class `VectorHeuristicSE`. Since this version of heuristic has a completely linear belief update process, all of the code – the belief state transformations as well as the combinations, shifts and extractions of the subprocesses states – can be and is implemented via matrix operations. These matrices are all computed in a constructor of the `VectorHeuristicSE` class and finally they are multiplied to form a single array of state estimator matrices for each action and observation, equivalent to the one used in `GenericStateEstimator`. The reward vectors are calculated similarly.

The interface of this class is made compatible with `GenericStateEstimator` as well, so when the incremental pruning algorithm is executed, an instance of this class is passed to the `IncPruner` object instead of a `GenericStateEstimator` instance.

Chapter 7

Results

The proposed heuristics rely in many ways on the fact that reasoning about the state in partially observable Markov decision processes is generally a convergent process and most of the errors that might arise while filtering the observations will decrease in time, as was thoroughly studied in [Boyen and Koller, 1998]. It would be hard to find and prove exact bounds for the error in general, and as the author believes, they would not necessarily characterize an average error on typical problems. It was therefore decided to rather demonstrate the performance of the heuristic statistically on individual examples.

There were several reasons for comparing the implementations only by accuracy of the result, ignoring times of execution. The main contribution of the proposed heuristic is that it cuts down the asymptotic complexity from superexponential to exponential, allowing to solve greater problem configurations that would be completely intractable within the exact algorithm. On the other hand, the implementations vary greatly in complexity constants. While the implementation of incremental pruning uses an optimized and compiled library for its most crucial task – pruning the vector sets, the grid approximation algorithm is written only in Python and NumPy and it is not optimized that much for performance. It turns out that for problems small enough to be solvable by all of the implementations, these constants often outweigh the asymptotic advantage of the heuristic and the execution times are rather similar.

7.1 Testing the implementations

The tests were performed on a Blade server with 16 CPU cores, which allowed for more evaluation runs and therefore lower statistical error of

the results. All of the configurations used discounted rewards in attempt to simulate an infinite process, so the evaluations were performed with more time steps than the number of learning time steps to better explore the behaviour of the resultant policy.

7.1.1 Incremental pruning

An extensive amount of problem configurations have been tested with the generic incremental pruning algorithm. As it turned out, the super-exponential complexity was a major obstacle: unless the parameters of the problem were fine–tuned by many trials and errors, almost all of the configurations either ended with a trivial result (only one vector in the value function set resulting in a constant choice of the same action, never picking up any fork) or the number of vectors in the value function set increased at a certain point of execution so rapidly that the algorithm was unable to even complete the current value iteration step and retrieve a consistent value function as a result. In successful cases the resultant value function representation usually contained hundreds or thousands of vectors. The algorithm was unable to continue whenever there appeared more than approximately 50 000 vectors in the set before pruning.

There are two experiments with incremental pruning presented here. Because of the necessary parameter tuning they don't make a nice example of the dining philosophers problem and serve rather as an illustration of the precision of the generic grid approximation algorithm, which will be fully exploited later in comparison with the heuristic algorithm. The biggest obstacle was the fact that it was impossible to set up a configuration with longer learning time that would force the waiter to plan longer ahead and pick up forks more often (the only way to make a long learning time configuration solvable was to set up the probabilities and rewards in such a way that made the "pick up" action totally unprofitable).

Experiment 1 (table 7.1) is an example of such a configuration with parameters fine–tuned to be solvable by incremental pruning. The reward for picking up the fork was usually the last parameter to be tuned with respect to the other parameters, in this case it needed to be specified very precisely (5 decimal places) for the algorithm to produce a non-trivial result within an acceptable time.

As we can see from the results, the optimal policy represented by the result of the incremental pruning algorithm was not much better than the constant "do nothing" policy in terms of average discounted reward. Both fixed grid approximation algorithms performed slightly worse than the optimal result, yet better than the constant policy. The heuristic

Numer of philosophers	3	
Probabilities	$p_{hungry} = 0.4$	$p_{satiated} = 0.6$
	$p_{drop\_fork} = 0.2$	
Rewards	$r_{cannot\_eat} = -1$	$r_{pick\_up} = -1.05595$
Discount factor	$\gamma = 0.9$	
Learning time	5 steps	
Evaluation time	256×40 steps	

Algorithm	Average reward	Pick up
Incremental pruning	-7.279	9 %
Generic fixed grid	-7.430	9 %
Heuristic fixed grid	-7.373	10 %
(never pick up)	-7.747	0 %
(always pick up fork 1)	-17.558	100 %

Table 7.1: Experiment 1 – non–trivial configuration

algorithm, despite of using much stronger approximation, turned out to be comparable or even slightly better than the generic approximation algorithm in this case. All of the non–constant policies resulted in approximately 1 "pick up" action for every 10 time steps.

An example of a configuration where the "pick up" actions were performed more often is experiment 2 (table 7.2). Unfortunately, such behaviour was only achievable with very shor learning time – only 3 time steps. Nevertheless, even for such a short time there was a parameter combination resulting in still not fully trivial results.

The fixed grid approximation algorithm performed almost as well as the exact algorithm also in this experiment. The heuristic was only slightly better than the best constant policy this time, however, this doesn't mean that the resulting policy was trivial, in fact it resulted in 33 % "pick up" actions as compared to 15 % for the generic algorithms. That also excludes any random behaviour as it would lead to much worse results, should there be a similar frequency of picking up forks.

7.1.2 Heuristic vs. generic fixed grid algorithm

Not using the incremental pruning algorithm, the complexity of finding a solution was only dependent on the process's "quantity" parameters (philosopher geometry, fineness of the value function grid, number of learn-

Numer of philosophers	3	
Probabilities	$p_{hungry} = 0.4$	$p_{satiated} = 0.6$
	$p_{drop\_fork} = 0.2$	
Rewards	$r_{cannot\_eat} = -1$	$r_{pick\_up} = -0.5$
Discount factor	$\gamma = 0.9$	
Learning time	3 steps	
Evaluation time	256×40 steps	

Algorithm	Average reward	Pick up
Incremental pruning	-6.672	15 %
Generic fixed grid	-6.862	14 %
Heuristic fixed grid	-7.475	33 %
(never pick up)	-7.566	0 %
(always pick up fork 1)	-11.807	100 %

Table 7.2: Experiment 2 – short learning time

ing steps) and not on the "quality" parameters (conditional probabilities, rewards) which could then be configured in a manner that demonstrated a more sophisticated reasoning.

Experiment 3 (table 7.3) was set up in a way to strongly force the waiter to pick up forks. A reward for picking up a fork was set ten times smaller than the reward for a ringing philosopher, making the important decision rather "what fork to pick up now" instead of "whether to pick up a fork at all". The philosophers were set to get hungry as well as drop forks more often and the learning time was increased to 40 steps. Most importantly, the discount factor was set to 0.99 not to prefer immediate rewards over long-time rewards.

In the results of this experiment, both algorithms performed quite similarly in terms of average reward, which was obviously much better than the reward of any constant policy. The interesting result was the actual behaviour of the policies: although the policy from the heuristic algorithm picked up forks much more often (96 % vs. 64 %), the average reward differed only slightly, probably because it chose to pick up forks even when less necessary, which didn't raise the cost too much, as was given by the parameters.

Experiment 4 (table 7.4) was an attempt to simulate a process more similar to the experiments with the incremental pruning (1 and 2), but to explore it more precisely, taking advantage of the possible longer learning

Numer of philosophers	3	
Probabilities	$p_{hungry} = 0.4$	$p_{satiated} = 0.6$
	$p_{drop\_fork} = 0.3$	
Rewards	$r_{cannot\_eat} = -1$	$r_{pick\_up} = -0.1$
Discount factor	$\gamma = 0.99$	
Learning time	40 steps	
Evaluation time	1024×80 steps	

Algorithm	Average reward	Pick up
Generic fixed grid	-37.980	64 %
Heuristic fixed grid	-38.655	96 %
(never pick up)	-63.388	0 %
(always pick up fork 1)	-60.931	100 %

Table 7.3: Experiment 3 – planning longer ahead

Numer of philosophers	3	
Probabilities	$p_{hungry} = 0.5$	$p_{satiated} = 0.8$
	$p_{drop\_fork} = 0.4$	
Rewards	$r_{cannot\_eat} = -1$	$r_{pick\_up} = -1$
Discount factor	$\gamma = 0.8$	
Learning time	40 steps	
Evaluation time	1024×80 steps	

Algorithm	Average reward	Pick up
Generic fixed grid	-4.939	5.3 %
Heuristic fixed grid	-4.919	0.3 %
(never pick up)	-4.966	0 %
(always pick up fork 1)	-9.452	100 %

Table 7.4: Experiment 4 – “picking up” less profitable

Numer of philosophers	4	
Probabilities	$p_{hungry} = 0.4$	$p_{satiated} = 0.7$
	$p_{drop\_fork} = 0.3$	
Rewards	$r_{cannot\_eat} = -1$	$r_{pick\_up} = -0.5$
Discount factor	$\gamma = 0.95$	
Learning time	20 steps	
Evaluation time	1024×80 steps	

Algorithm	Average reward	Pick up
Generic fixed grid	-20.569	33 %
Heuristic fixed grid	-22.356	60 %
(never pick up)	-25.819	0 %
(always pick up fork 1)	-34.264	100 %

Table 7.5: Experiment 5 – bigger system

and evaluating times. Setting the discount factor to a value of 0.8 made long–term rewards much less profitable (with the learning time of 40 steps and since $0.8^{40} \approx 0.0001$, this was nearly a perfect approximation of an infinite horizon). The probabilities of philosophers getting hungry, finishing eating and dropping forks were raised to make the whole process faster to accommodate the lower discount factor.

As could be expected, the average rewards for the algorithm generated policies were not much better than the constant "never pick up" policy. The heuristic algorithm was a little surprise here as it not only performed slightly better than the generic algorithm, but it chose to pick up forks much less often (0.3 %) than the generic algorithm (5.3 %), even though in all of the other experiments it tended to pick up more often.

The last experiment (table 7.5) used a bigger configuration, 4 philosophers instead of 3. The fineness of the value function grid was lowered from 4 points per dimension to only 3 points. Though it might seem that such a coarse grid would not be able to represent any non–trivial policy, in fact it was. Other parameters were set to ordinary values. In this experiment the heuristic algorithm policy chose to pick up more often again and it performed worse than the generic algorithm policy, yet still noticeably better than the constant policies.

7.1.3 Modified vector heuristic

The author was unable to find a combination of parameters that would lead to nontrivial yet computable results with the vector based heuristic implementation. It is however possible that further experiments might produce some results, as some of the mid-computation values indicated a non-trivial and non-random behaviour.

7.2 Discussion

In the first part of this work, an introduction to dynamic troubleshooting, Markov decision processes and partially observable Markov decision processes has been provided. Some widely used algorithms have been presented to give the reader a notion about solving POMDPs.

In the second part of the work, a specific example of a POMDP problem has been introduced and the methods of its solution have been explored. A heuristic that was new or at least unknown to the author at the time of writing has been proposed for the problem and a way to generalise the heuristic to more POMDP problems has been suggested. A considerable amount of software has been programmed – the problem abstraction, the generic algorithms as a comparison and the heuristic itself.

The heuristic in its first proposed form has been extensively tested on the presented problem. The results have proven that, even though the heuristic uses much stronger approximation compared to the generic algorithm and as a result of completely different time complexity it allows to solve problems that exceed the capabilities of the generic algorithms, it performs comparably on the presented problem and it provides an opportunity to be further used on similar problems, generalised, improved or possibly used with other attitudes to storing value function.

What remains unknown to the author at this time is the usefulness of the second proposed form of the heuristic (the vector variation). There is still a possibility that it could be used more appropriately and produce some useful results.

Bibliography

[Bellman, 1957] Bellman, R. E. (1957). *Dynamic programming*. Princeton University Press, Princeton, NJ, USA.

[Boyen and Koller, 1998] Boyen, X. and Koller, D. (1998). Tractable inference for complex stochastic processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 33–42.

[Cassandra, 1998] Cassandra, A. R. (1998). *Exact and approximate algorithms for partially observable markov decision processes*. PhD thesis, Brown University, Providence, RI, USA. Adviser-Kaelbling, Leslie Pack.

[Cassandra, 2005] Cassandra, A. R. (2005). pomdp-solve. Computer program, http://www.pomdp.org/pomdp/code/index.shtml.

[Cassandra et al., 1994] Cassandra, A. R., Kaelbling, L. P., and Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1023–1028, Seattle, Washington, USA. AAAI Press/MIT Press.

[Cassandra et al., 1997] Cassandra, A. R., Littman, M. L., and Zhang, N. L. (1997). Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *In Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 54–61. Morgan Kaufmann Publishers.

[Dempster et al., 1977] Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38.

[Howard, 1960] Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. Massachusetts Institute of Technology Press, Cambridge, MA, USA.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237.

[Ledl, 2004] Ledl, S. (2004). Dining philosophers - simulace víceprocesorového systému s možností selhání a oprav. Bachelor's thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

[Lovejoy, 1991a] Lovejoy, W. S. (1991a). Computationally feasible bounds for partially observed markov decision processes. *Oper. Res.*, 39(1):162–175.

[Lovejoy, 1991b] Lovejoy, W. S. (1991b). A survey of algorithmic methods for partially observed markov decision processes. *Annals of Operations Research*, 28(1-4):47–66.

[Monahan, 1982] Monahan, G. E. (1982). A survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16.

[Puterman, 1994] Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, Inc., New York, NY, USA.

[Puterman and Shin, 1978] Puterman, M. L. and Shin, M. C. (1978). Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11):1127–1137.

[Russell and Norvig, 2003] Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach, Second Edition*. Prentice Hall Pearson Education, Upper Saddle River, NJ, USA.

[Sallans, 2002] Sallans, B. A. (2002). *Reinforcement learning for factored markov decision processes*. PhD thesis, University of Toronto, Toronto, Ont., Canada.

[Sondik, 1971] Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, Stanford, California, USA.

[Sondik, 1978] Sondik, E. J. (1978). The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2):282–304.

[Sutton, 1996] Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press.

[Watkins, 1989] Watkins, C. H. C. J. (1989). *Learning from Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, UK.